NASA CONTRACTOR REPORT 166581

Guidelines for Testing and Release Procedures

Russell Molari and Mel Conway

CONTRACT NAS2-11555

April, 1984

NASA

NASA CONTRACTOR REPORT 166581

Guidelines for Testing and Release Procedures

Russell Molari and Mel Conway

Prepared for
Ames Research Center
under Contract NAS2-11555

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS
Continued

Section 0
SCOPE

This document contains guidelines and procedures for testing and release of computer software at NASA/Ames Research Center. The guidelines and specific recommendations are specialized to the type of software environments at Ames. Page 4-20 of this document is a simple form to use for more information or for sending comments.

REVISION LEVEL

This document is version 2.0. It is a complete revision from version 1.0, re-organized for more extensive content.

Section 1
INTRODUCTION

In a sense, all programs are created equal: they have at least some errors in them. Since all programs have to run correctly, those errors have to be removed during testing. This document provides guidelines for the most efficient and useful testing techniques in our software environments at Ames.

1.1 DO THE TESTING THAT FITS THE PROGRAM

This is the single most important rule to follow in software testing. Testing is a mixture of both science and art. It consumes between 30% and 50% of a project's software costs. (See Myers and Boehm in Section 6 references) Testing will be efficient, effective and satisfying if an approach is carefully selected and creatively adapted for each specific environment. This is especially true at Ames, where software project environments vary considerably, and often are highly research-oriented.

Some of the areas in which testing approaches must be selected or specialized for each environment are:

- When and how each of the four most common testing phases (unit testing, component integration testing, system integration testing, and acceptance testing) will fit into the project's development activities;
- When to write a test plan, who should write it, how detailed to make it, and how to review it;
- Which approaches to use in each testing phase, who should perform them, and how they should be reviewed;
- How to avoid over-testing or under-testing in each phase;
- Types of non-machine review to perform;
- Which test tools to use or develop;
- How to test in difficult environments, such as real time, multi-tasking, distributed networks, or simulations.

A wealth of excellent general advice is in the two classic texts

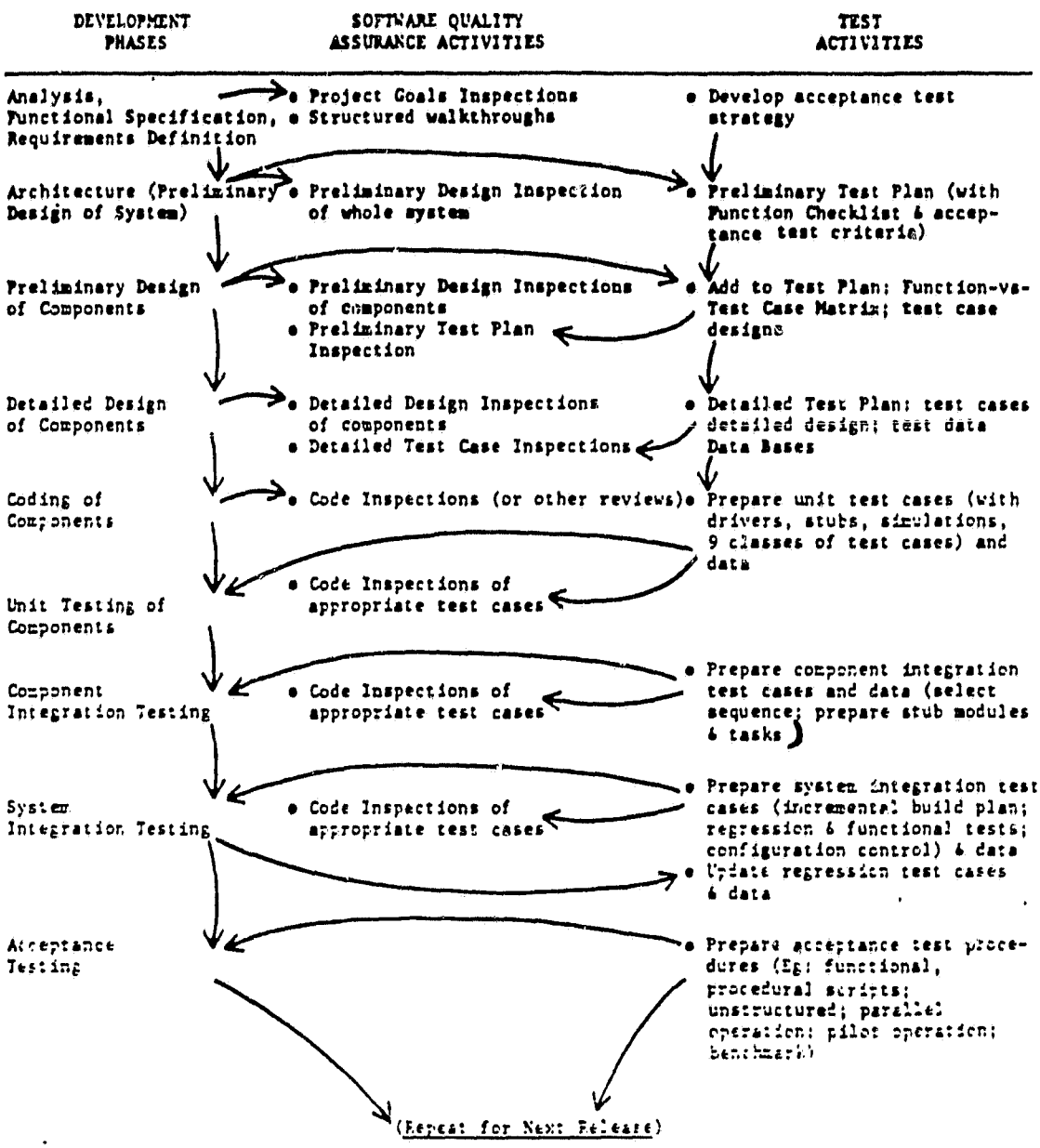
on testing by Myers (see references). In the context of such general advice, this document offers guidelines for specializing and selecting test approaches in all of the above concerns.

1.2 HOW TEST ACTIVITIES RELATE TO OTHER DEVELOPMENT AND RELEASE ACTIVITIES

Testing requires pre-planning, documentation, test data and test case preparation, test execution and debugging, reporting, and error correction.

The pre-planning is always important: without pre-planning and writing a good test plan, it can almost be guaranteed that later test activities will appear and feel like schedule overruns. All test activities should be scheduled into the development plans. An early, written test plan should outline the phases of testing which will be performed, and the approaches for each phase. From such a test plan, all activities relating to testing can then be planned and scheduled. The accompanying diagram shows the approximate relation between a full set of test activities and a full set of development and software quality assurance activities.

OUTLINE OF TEST ACTIVITIES IN RELATION TO OTHER PROJECT DEVELOPMENT WORK



1.2.1 The Bigger Projects

On a project which is large enough to require a team of several programmers, or long enough to span more than a year of development, the test activities may encompass the full set of activities shown in the diagram of Section 1.2. As the software design progresses from functional specification to detailed design of components, the test plan is developed from an acceptance test strategy into a set of test case and test data detailed designs. Four phases of testing are then performed, including complete system integration regression testing, formal acceptance testing, and maintenance of the test plans, data, and tools in anticipation of the next release.

In an environment such as this it is very efficient if a portion of the team specializes in the test activities, and performs their work independently and in parallel with the developers. Experienced, effective personnel are required for both types of work.

1.2.2 The Smaller Projects

On a project which requires only one or two programmers, or is only a few months of development, the test activities may be best if they are combined and simplified from the full set shown in the diagram. The test plan may be included with the software design documents. Inspections may be performed as Desk Inspections or with the help of personnel outside the task for critical parts. Component integration testing and system integration testing commonly become a single test phase on smaller projects, because of the smaller number of components. Acceptance testing may be less formal because of a closer, more continuous involvement of the Task Requestor or user.

Even though these environments cannot usually support a separate group of test personnel as in the bigger projects, an effective approach is to obtain outside help in reviewing and/or integration testing the most critical components or functions. These should be selected early with the Task Requestor or user, and should concentrate on the components with heavy I/O or hardware usage, complicated program interfaces, or critical user interfaces. This is a very effective compromise which should be used whenever possible on the smaller projects.

Section 2
GUIDELINES FOR DEVELOPING A TEST PLAN

The test plan defines what the test activities will be and how to prepare and execute them. It is developed in the following sequence:

(1) Define what phases of testing will be performed in relation to the project's specific design components and development plans.

(2) Outline acceptance criteria for the entire system, by preparing a function checklist for the whole system and developing acceptance criteria for each function. Define what features of the system, if any, will not be tested.

(3) Prepare the table of contents for the remainder of the test plan, and define when the various sections of details will be added. Define what personnel will perform the testing, and their relationship to the development staff. Define what personnel will review or inspect the test plan and test cases, and with what methods.

(4) Briefly outline the strategies, approaches, and general test data specifications for each testing phase in a "top-down" sequence (i.e., acceptance testing first, system integration testing next, component integration testing next, unit testing last).

(5) Develop and add the detailed procedures, test cases, and test data for each testing phase in a "bottom-up" sequence (i.e., unit testing first, etc.).

For each phase of testing, the test plan should focus on the planning and scheduling of the following activities:

- Gathering and generating of test data
- Establishing test libraries
- Establishing test data bases
- Choosing test tools
- Scheduling test resources
- Executing tests
- Analyzing test results
- Documenting test results

For each phase of testing, the test plan should also define all of that phase's test activities, their purpose, deliverables, responsible parties, procedures, and schedules.

2.1 WHEN TO DEVELOP THE TEST PLAN

The test plan evolves over the whole software development cycle. At each development phase, it is updated with more detail. The accompanying chart outlines the details which are added into the test plan at each development phase.

Test Plan Development

Software Development Phase	Details Added to Test Plan
Analysis and Functional Specification	• Definition of testing phases in relation to specific project plans • System function checklist • System acceptance criteria • Schedules and responsibilities
Architecture (Preliminary Design) of System	• Detailed acceptance criteria tied to requirements • Approaches and test cases for acceptance testing and system integration testing • Choice of test tools
Preliminary Design of Components	• Approaches and test cases for component integration testing • Function-vs-test case matrix • Details for all test cases already defined • Preliminary design of all new test tools
Detailed Design of Components	• Approaches and test cases for unit testing • Detailed design of all new test tools • Plans for test libraries • Plans for test Data Bases • Scheduling of test resources
Coding and Unit Testing	• Code and/or documentation of unit test cases and data and all new test tools • Documentation of test results
Component Integration Testing	• Code and/or documentation for component integration test cases and data • Documentation of test results

System Integration Testing	• Code and/or documentation for system integration test cases and data • Documentation of regression test procedures • Revised function-vs-test case matrix • Documentation of test results
<u>Software Development Phase</u>	<u>Details Added to Test Plan</u>
Acceptance Testing	• Test scripts and documentation of test cases and data for acceptance testing • Documentation of test results

2.2 WHO SHOULD BUILD THE TEST PLAN

The personnel who have primary responsibility for a test effort should write the corresponding portions of the test plan. As described in Section 1.2, overall test responsibility may range from an independent test group for larger projects, to the development team members or users for smaller projects. In all cases, the Task Manager should also be closely involved in building the test plans. In general, the users should contribute the most to the test cases and test data of the higher phases of testing (acceptance testing and system integration testing) and development team members mostly to the lower phases. Since acceptance test criteria are the first items to be written into the test plan and the last to be implemented, users should be involved at the earliest and the latest stages of the test plan, with the development members being the heaviest involved in the middle stages.

2.3 AN OUTLINE FOR AN EFFECTIVE TEST PLAN

2.3.1 General Outline

The test plan contains a general set of information about the overall system--wide test approach, as well as a detailed specification for each phase of testing to be performed. A possible outline follows.

(A) PURPOSE AND SCOPE

Define the purpose, coverage and limitations of the test plan. Identify the organizations and personnel to whom the requirements of the test plan apply. List the important existing documents which relate to the design and testing requirements for the system.

(B) OVERALL TEST PHILOSOPHY

Select the phases of testing which will be performed (i.e., the four main phases described in these guidelines) and state them in terms of the project's specific design and development plan. For example, if two independent subsystems are to be developed and then integrated, the project's testing may consist of all four phases separately for each subsystem, followed by separate component integration testing, system testing, and acceptance testing for the combined system--a total of 4 + 4 + 3 = 11 test phases in all.

(C) OVERALL COMPUTING ENVIRONMENT

Describe the hardware and software computing environment for the system as a whole, including all packages and devices to which the system interfaces.

(D) OVERALL TESTING SCHEDULE

Define when each section of the test plan will be added in full detail. Define when each testing phase is expected to begin and end, and the schedules for all known testing activities. Schedules should relate to the development schedule when appropriate.

(E) SPECIFICATIONS FOR EACH PHASE OF TESTING

For each of the testing phases identified in (B), the following is added in stages, as described in Section 2.1:

- Purpose

- Entrance and exit criteria for the phase
- Dependencies which must be satisfied before the phase can begin
- A brief description of the testing phase and the approaches selected to accomplish it
- A description of the specific computing environment for this phase, and any resources required
  - Operating system environment, any required packages, and any related software upon which the system or data depends
  - Personnel responsibilities and requirements, including those responsible for coordinating and conducting the tests, those required to prepare test drivers and inputs, those required to evaluate the outputs, and those required to operate equipment.
  - Any training which is required for the developers, testers, or operators
  - The computers and related equipment which are required during the testing phase, plus the required operating time and availability for each.
- A list functions or components which will not be tested in this phase.
- The test cases to be developed or run, and for each:
  - Purpose
  - Which functions in the Function List they will test
  - Program logic for a program, or format and contents for test case data
  - Expected results

  In preliminary versions of the test plan, this may be simply an estimate of the number of test cases of each type.
- Test tools which will be required for this phase of testing. If they are to be developed, include their design as for any other program. If they are to be acquired, include their source, schedule, and how they should be tested.
- The deliverables and methods for recording results and reporting and correcting problems. In certain phases, problems may be corrected directly by the tester. Recording of the results via test logs is always desirable. In some phases, a formal written procedure is used for reporting a problem and submitting the report for correction by other personnel. The procedures should be explicitly stated, even if informal.
- The sign-off method should be described, even if informal. This should define which personnel (e.g., librarian, Task Manager, Chief Programmer, Task Requestor) should approve each of the test results and allow further testing phases to begin.
- Any risks associated with this phase of testing

should be estimated. This may include estimated schedule impacts if certain fundamental problems are discovered or if machines are unavailable, etc.

(F) TEST TOOLS

Describe or design any test tools which will be required throughout most of the testing phases, even if they already exist in the system. Describe the source, schedule, and testing of tools to be acquired. Include the logic and design of tools to be developed. List any tools which already exist in the system or operating system, to call them to the attention of the test personnel.

(G) FUNCTION CHECKLIST
See Section 2.3.2.

(H) FUNCTION-VS-TEST CASE MATRIX
See Section 2.3.3.

2.3.2 Function Checklist

One of the simplest, most useful tools to support testing is a terse, comprehensive list of all of the system functions. This list can be used as a checklist for developing and executing test cases and test procedures. It is also the basis for the function-vs-test case matrix, described in Section 2.3.3.

A slightly more sophisticated function checklist may be used which also incorporates a matrix identifying which executable programs support each function. This "function-vs-component" matrix can be used to help design test cases, as well as to facilitate tracking of the software as it is developed.

2.3.3 Function-Versus-Test Case Matrix

The function-vs-test case matrix provides an additional dimension to the function checklist. For each function within the system, this matrix identifies which test cases will test the function. A different matrix may be required for system integration testing and for acceptance testing, and often even for specific approaches within each of these pahses (e.g., a matrix specifically to identify regression test cases).

The accompanying chart illustrates a very simple example of this matrix, as well as the function checklists described in Section 2.3.2. In this example of a hypothetical inventory management system, a user can inquire, change, add, or delete a part or assembly. The system is composed of only three executable programs, performing inquiries, add-or-delete, and change. Only four test cases are used.

A SIMPLE FUNCTION CHECKLIST

1. INQUIRY OF AN ASSEMBLY
2. INQUIRY OF A PART
3. ADD AN ASSEMBLY
4. CHANGE AN ASSEMBLY
5. DELETE AN ASSEMBLY
6. ADD A PART
7. CHANGE A PART
8. DELETE A PART

A FUNCTION-VS-COMPONENT MATRIX

FUNCTION	PROGRAM 'INQUIR'	PROGRAM 'ADDDEL'	PROGRAM 'CHANGE'
1.	X		
2.	X		
3.		X	
4.			X
5.		X	
6.		X	
7.			X
8.		X	

A FUNCTION-VS-TEST CASE MATRIX

FUNCTION	CASE ASSEMB1	CASE ASSEMB2	CASE PART 1	CASE PART 2
1.	X			
2.			X	
3.		X		
4.	X			
5.		X		
6.				X
7.			X	
8.				X

EXAMPLES OF FUNCTION CHECKLISTS AND MATRICES

2.4 TEST PLAN AND TEST CASE INSPECTIONS

Whenever possible, an Inspecation should be held to review a system's test plan and its test cases. The Guidelines for Software Inspections (see references) contains Criteria for Materials and a Checklist for conducting these Inspections. The Checklist covers five categories: specifications and overall requirements, strategy, tools, functional coverage, and test cases. The Inspection is designed to uncover problems related to insufficient planning, improper or unnecessary tools, potential over-testing or under-testing, and inappropriate approaches.

A test plan can be reviewed in a Test Plan Inspection in this way during its preliminary stage, before test cases are fully designed. A Test Case Inspection can then be conducted later, when the test cases have been designed, by using the same checklist. In many cases, however, a single Test Plan/Test Case Inspection is sufficient.

Section 3
GUIDELINES FOR SELECTING AN OVERALL TEST APPROACH

In order to determine the best overall testing approach, the most important considerations are:

- Application
- Size of staff
- Release cycle

3.1 APPLICATION

The application of the software system often dictates the types of testing which should be emphasized. At all stages of testing a common-sense selection can be made which will optimize testing for a specific application. Some guidelines include:

- For interactive, data entry, or editing applications, emphasize system integration testing and acceptance testing.
- For data basing applications and computational systems, emphasize complete test case coverage at the system integration testing phase, and during that phase's regression testing (See Section 4.3).
- For a real time I/O, control, or data acquisition system, emphasize unit testing; include performance testing during the integration testing phases.
- For graphics systems, emphasize unit testing with full test case coverage, and full acceptance testing.
- For software supplying or making heavy use of operating system services, emphasize complete test case coverage at all lower phases of testing.

3.2 SIZE OF STAFF

For a Task with a small staff, the Task Manager or even the Task Requestor will have to take an active role in the testing process. This environment may also utilize temporary help from outside the Task. Since such personnel may be less familiar with the system's design that the developers, they should primarily contribute at the system integration and acceptance testing phases. Since they will be less available, the most critical functional tests should be selected for them to prepare and run, to make the best use of their effort. When project staffing is small, reporting of the testing results should also be emphasized, to allow the Task Requestor to review them as time permits.

For projects with a larger staff, the team can be sectioned into an independent test group. Alternatively, the staff's assignments can be arranged so that each team member performs the testing of other members' components. A sufficient number of users will be available to involve them in the appropriate activities during system integration testing and acceptance testing. Inspections of both the software and the test plan should be possible. Configuration control will be a critical element during testing, and a librarian should be designated. Component integration testing should be performed using incremental builds.

3.3 RELEASE CYCLE

Software projects span a range of expected life cycles, such as:

(1) Some projects, generally the smaller ones, are developed in a life cycle consisting of: analysis--design steps--testing phases--release to user. The system or program is intended to perform its function for a short lifetime, or it has a limited and well-defined functional specification. The software is released once, and not planned for further release.

(2) Some projects are larger or have more complex functionality, and are developed in a life cycle consisting of: analysis--system design--design, testing, and release of a subset of components--design, testing, and release of a larger subset--etc. until all components are released. This is a planned evolutionary approach which releases the system to the users in increasingly more complete versions.

(3) Some projects are developed for long lifetimes, and have a life cycle consisting of: analysis--system design--design, testing, and release of a baseline version--design, testing, and release of a revised version containing enhancements--etc. throughout the lifetime. This approach is planned for change, accounting for functional, user, and technology requirements changes.

If a single release environment such as (1) is planned, testing should consist of the phases and approaches which are most applicable, but generally not utilizing regression test methods during system integration testing. These environments should utilize a minimum number of tools. Test cases and results should be documented while being performed, but need not be saved in all cases.

When multiple releases, such as (2) or (3), are planned for a

system, a more rigorous set of test activities should be utilized. Test approaches, configuration control, and reporting are all more elaborate in these cases, in order to:

- Rigorously test system enhancements in each release.
- Ensure that each enhancement does not cause rippling malfunctions throughout the system.
- Ensure that the enhancements do not adversely impact the "unchanged" portions of the system.
- Maintain user satisfaction and confidence.

A sample scenario of such a multi-release testing cycle follows:

(a) The system is first developed and released through all four phases of testing and accepted by the user. This first release is copied and used as the production baseline. The key system integration test cases and results are stored as regression test cases for further use.

(b) Enhancements for the next release are specified and designed. The test plan is udpated:

- New functions are added to the function checklist.
- The function-vs-test case matrix is reviewed to eliminate irrelevant cases or specify new ones.
- New test cases are designed.

(c) A copy of the baseline is made to begin development of the next release. The enhancements and new test cases are developed. Unit testing and component integration testing are performed.

(d) All new test cases for system integration testing are run on the system under development. These test cases and results are saved.

(e) A second copy of the baseline is made for regression testing. Regression tests from the previous release are run on the system under development and compared to previous results. If necessary, these regression test cases may be run on the second "regression" copy, with minor modifications to either the software or the test cases, in order to allow direct comparison with results from the system under development. All regression test cases and results are saved.

(f) User-oriented portions of system integration testing

are performed. Independent test teams are used whenever possible.

(g) A copy of the system under development is made in its production format and used for acceptance testing. Changes resulting from this acceptance testing are returned to the system under development. Cutover is scheduled with all parties involved.

(h) Production media are produced for the new release. At least one copy of the production software is saved as the new baseline of the system.

(i) The new system integration test cases and results used in step (d) are selectively merged with the regression test cases and results used in step (e), to create an updated set of regression test for this release.

(j) A copy of the new baseline is made for investigating any future production problems. Any immediate corrections required during the baseline's lifetime may be tested with the updated set of regression tests (i) for this release. These corrections should be carefully reported, and manually merged into any new development work for the subsequent release.

(k) New enhancements for the next release begin again at step (b).

The specific system and project environment will further dictate detailed configuration control procedures, backups to be made, etc.

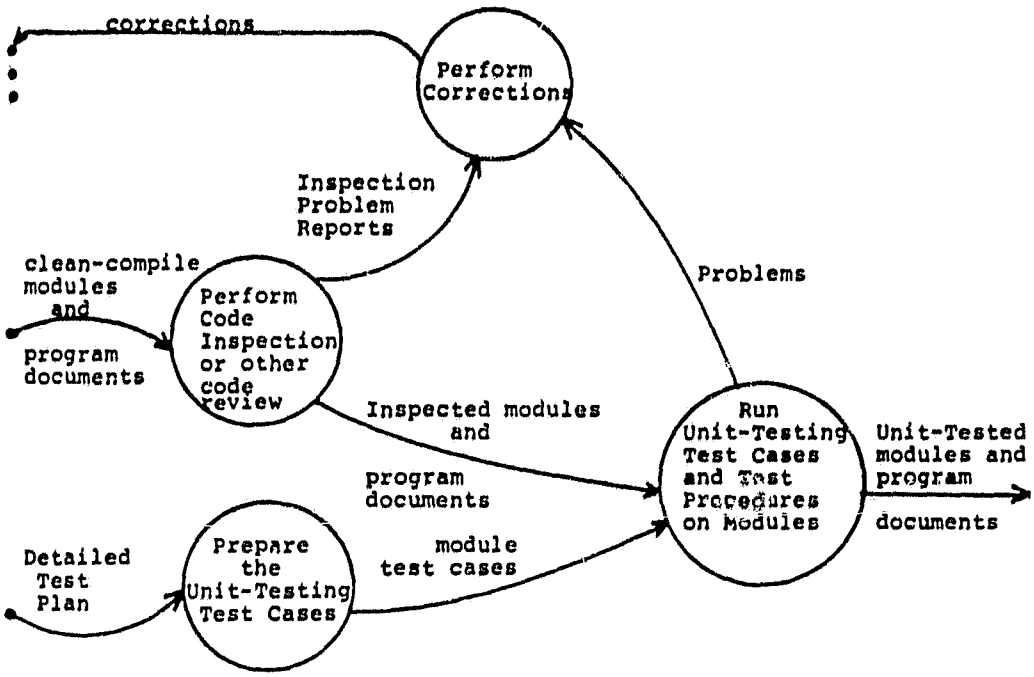
Section 4
GUIDELINES FOR EACH PHASE OF TESTING

For each phase of testing which a set of software undergoes, there are options as to which procedures will be best for a specific environment. Regardless of the procedures selected, each phase of testing still has a well-defined purpose, as well as a definition of what is tested and what it looks like after it is tested.

The following sections provide specifications and guidelines for each phase of testing, covering:

- Purpose of this phase of testing
- Entrance criteria
- Exit criteria
- Personnel performing this phase of testing
- Methods, and how to select them
- Tools which can be used
- Guidelines pertaining to some specific software environments.

4.1 A GENERAL SPECIFICATION FOR UNIT TESTING OF COMPONENTS



4.1.1 Purpose

Unit-testing verifies that isolated module(s) perform as specified in functional specifications and program documents.

Each new or modified module is unit tested in isolation before being tested with the rest of the system. This is the stage in which to exercise all the paths through the modules, since some paths are difficult to generate in a realistic situation.

4.1.2 Entrance Criteria

- Modules have a clean compile
- Modules have been submitted for (preferably passed) a Code Inspection, Code Desk Inspection, or other code review
- Program documents associated with the modules are up-to-date
- If the modules to be tested have any logical dependencies on other primitive modules, those should have already passed unit-testing
- A development system is available with compiler, development tools, unit testing tools, and any hardware devices required to test the modules
- The Detailed Test Plan for these modules is up-to-date

4.1.3 Exit Criteria

- The modules pass all logic and functionality tests and test cases, according to the procedures selected from Section 4.1.5
- Program documents relating to the modules are updated
- Code Inspection (or other code review) is passed
- If the project has a librarian, modules are signed off by the librarian

4.1.4 Personnel

At the module level, unit testing is generally the responsibility of the programmer. In cases where more than one programmer develops a program, the best choice is for the most senior person who did not write a module to be the one to unit test it. In the case of utility modules, it is advantageous for unit testing to

be performed by another unrelated "tester".

4.1.5 Unit Testing Methods

4.1.5.1 General Description

The steps involved in unit testing are generally as follows:

(1) Develop the modules' code, according to its detailed design, until a clean compile is obtained.

(2) Perform review (Code Inspection, Code Desk Inspection, Structured Walkthrough, etc., as outlined below)

(3) Select a sequence in which the modules will be tested.

(4) Select test cases to be performed on each module, and a method from which to run them.

(5) Prepare test case data and run the test cases.

(6) Perform corrections and re-test.

(7) Maintain a test log; obtain signoff from librarian or designated personnel.

4.1.5.2 Methods and How to Select Them

The level of detail at which to unit test should vary according to how critical, how widespread, or how complex a module's function is. Complete unit-testing should:

- Exercise all code in the module and all parameter cases.
- If the program requires operator interactions or input, obtain independent opinions as to whether or not it is appropriate and understandable.
- Cause each error message (if any) to be written (to a spooled file or a CRT). Be sure they are meaningful and helpful; always obtain independent opinions regarding this.
- Execute all loops the minimum and maximum number of times possible.
- Verify that any files written or fields set in shared memory areas by the modules are correct according to the input given them. In many cases, local commons should be dumped with special debug code.

- Verify that any external flag modification is done correctly.

Possible methods to select among for testing individual modules follow.

Isolate and Completely Test with a Driver
For utilities, modules which will be utilized by many programs (such as a file open utility), and modules which serve critical functions (such as a module controlling a device in real time), unit testing should be accomplished by isolating the module and calling it from a specially-written test driver. Test cases should cover all aspects of the module's logic and performance.

Isolate and Moderately Test with a Driver
For complex modules which are not widespread utilities or which are used in very restricted environments, unit testing should also be done through isolating the module and calling from a test driver. However, the coverage of the test cases need not be totally complete. Some parameter test cases and logic path exercises can and should be substantially reduced.

Test from the Calling Program
Except when the overall unit-testing sequence is chosen to utilize the above "driver" methods, a module which is less complex and which is utilized in one or very few locations can be tested entirely from the program or module which calls it. The calling program can be externally manipulated (if an interactive debugger is available) or temporarily modified to perform a moderately complete selection of test cases. This is the best choice for most computational programs and most applications which do not perform critical I/O, data basing, or device manipulation. This method is otherwise preferable because it minimizes the coding of special test drivers, and it embeds the modules in the same environment during testing that they will have later during operation.

Inspections, code reviews, and desk checking
Regardless of which of the above methods is selected, some type of non-machine review should always be done.

- On large projects, Code Inspections or Code Desk Inspections are strongly recommended.

- For modules performing heavy user interaction or which interface to other incomplete software subsystems or hardware devices, less formal structured walkthroughs or code reviews are recommended, attended by the personnel implementing the related subsystem or hardware.

- For small projects or one-person tasks, the most critical modules should also be reviewed in a structured walkthrough with the Task Requestor. If the modules are complex, it is highly recommended to perform a formal Code Inspection or Code Desk Inspection by borrowing temporary assistance from programmers on other tasks, with the Task Requestors' permission.

- For less critical modules on small projects or one-person tasks, the programmer (or a cooperative Task Requestor) should conscientiously desk-check the code prior to machine testing. Desk checking should be done with the appropriate language Code Checklist from the Guidelines for Inspections.

Non-machine checking of code is consistently a highly cost effective use of time. Problems detected at a programmer's desk can cost 10 to 50 times less to detect and correct than using machine testing.

Simulation

If a software subsystem (e.g., data base or other package) or hardware device is not available at unit test time, the unit testing should be postponed whenever possible. Constructing simulation software is always costly and usually only partially effective. Simulation software should be written only for extremely critical schedules. Whenever possible, the modules simulating a software or hardware sybsystem should be reviewed with personnel familiar with the ultimate subsystem. The simulation modules should allow for error conditions to occur, and should undergo Code Inspections to prevent large efforts to debug the simulation. Hardware devices require the simulation modules to include timing, and they should therefore be performance tested to measure their actual effective timing.

4.1.5.3 Guidelines for Selection of Unit Test Cases

The parameter cases to be used for unit testing a module can be determined by using the list below, which defines increasingly rigid categories of unit test cases. Proceed down the list, writing down all the input sets necessary to satisfy each step. For most modules, many of the steps will be redundant. Only those input sets which are unique need then be prepared and executed, but all of the purposes of each test must still be checked when the output is examined.

1. Parameter limits: Input each parameter at its upper and lower value limits. Input parameter sets which will cause each output value to occur at its upper and lower limits.

2. Parameter partitioning: Input one value of each

parameter to represent each of the various ranges ("partitions") it may take on. For example, for a full-range integer parameter, include the negative range, positive range, and zero. For a character parameter, include a letter, a digit, a special symbol, and a blank. Similarly, input parameter sets to test the various partitions of each output value.

3. Return codes: Input a parameter set which will cause each possible return or status code of the module to occur. In some cases, this will require an environment to be produced by the test driver. Operating system service calls should be simulated to produce various return codes only if it is critical to the module's operation. In most cases, only the environments which can be produced externally (e.g., device not ready, subsystem not installed, driver not loaded, etc.) need be tested.

4. Messages: Input parameter sets to cause each possible message in the module to be tested once.

5. Isolated failure modes: Produce external environments which contain single isolated error-causing problems (such as tape not mounted, file missing or locked, etc.) to test the proper reaction of the module. (See Appendix C for further description).

6. Interactive mistreatment: Modules which are interactive should be deliberately tested with all possible illegal action on the part of the user. (E.g., number instead of letter input, carriage return with no input, etc.).

7. Linkages: Input parameter sets which will cause the module to call every sub-program which it references.

8. I/O: Input parameter sets which will cause the module to execute every I/O, READ, WRITE, OPEN, etc. which it contains. FORMATS should be tested in each of their partition ranges. Test each file for: I/O of the first and last records, missing file and empty file.

9. Logic paths: Input parameter sets which will cause the module to execute every test-and/or-branch path at least once.

4.1.5.4 An Example of Selection of Unit Test Cases

For unit testing the following hypothetical FORTRAN module (header and comments are abbreviated):

```
      SUBROUTINE IVOUT (IVEC,LENGTH,LUNIT,IERR)
C
C PRINT AN INTEGER VECTOR IF IT IS NON-ZERO
C
C         IVEC-  INPUT-THE VECTOR
C         LENGTH-INPUT-LENGTH OF THE VECTOR
C         LUNIT- INPUT-PRINT OUTPUT LOGICAL UNIT NO.
C         IERR-  OUTPUT-RETURN STATUS:
C             0=SUCCESS
C            -1=ILLEGAL LENGTH INPUT
C            -2=ILLEGAL LUNIT INPUT
C
      DIMENSION IVEC (LENGTH)
C
      IERR=0
      IF (LENGTH .LE.0)  IERR=-1
      IF (LUNIT .LE.0)  IERR=-2
      IF (IERR .NE.0)  GO TO 999
C
      DO 1 I=1, LENGTH
         IF (IVEC(I).NE.0)  GO TO 10
    1 CONTINUE
C
C VECTOR IS ALL ZERO;  RETURN
C
      GO TO 999
C
C PRINT
C
   10 WRITE (LUNIT,500)  IVEC
C
  500 FORMAT(/, ' NON-ZERO VECTOR',/, (518))
C
  999 RETURN
      END
```

Proceeding through the unit test checklist suggests the following cases (for each case, the various suggested parameter values are listed, separated by semicolons:

```
Class 1:  LENGTH = 1; D (= highest dimension module can be
                    expected to handle)
          LUNIT = 1; U (= highest logical unit which can be
                    expected)
          IVEC = +-32767,...
```

```
Class 2:  LENGTH = 5
          LUNIT = 5
          IVEC = 5 * -5      ;     5 * 0      ;
                 5 * 5       ;    -5,0,5,-5   ;
                 (tests all legal partition ranges)

Class 3:  LENGTH = -1; 0          (produces IERR = -1)
          LUNIT = -1; 0           (produces IERR = -2)
          LENGTH = 5              (produces IERR = 0)
          LUNIT = 5               (produces IERR = 0)

Class 4:  None

Class 5:  Place printer off-line; unload printer driver;
          assign LUNIT to illegal device.

Class 6:  None

Class 7:  None

Class 8:  LENGTH = 1      ;   5          ;  7
          LUNIT = 1       ;   5          ;  5
          IVEC = -5       ; -5,0,5,0,-5  ; -5,0,5,0,-5,0,5
          (Tests WRITE statement for various partition
           ranges of the FORMAT (5I8).)

Class 9:  LENGTH = 5; 0           (test first "IF")
          LUNIT = 5;  0           (tests second "IF")
          IVEC = 0,0,0,0,0        ;5,5,5,5,5  (tests third "IF")
```

Duplicate test cases should now be eliminated. The actual test cases to be used should be the unique subset of the above cases, and are summarized in a test case matrix as follows:

INPUT TEST CASE	LENGTH	LUNIT	IVEC	CLASSES TESTED
1.	-1	1	5	3
2.	0	1	5	3
3.	1	-1	5	3
4.	1	0	5	3
5.	D	U	D*5	1
6.	7	5	7*-5	8,2,3
7.	5	5	5*0	2,3,5
8.	5	5	5*5	2,3
9.	5	5	-5,0,5,0,-5	2,3
10.	5	5	5*-32767	1
11.	5	5	5*32767	1
12.	Case (11) with printer off-line			5
13.	Case (11) with LP driver unloaded			5
14.	Case (11) with LUNIT assigned to CR:			5

4.1.6 Tools for Unit Testing

To perform complete coverage unit testing, module(s) must be executed in a number of fixed environments to test internal operation and outputs. For some of the methods outlined in Section 4.1.5, this requires a "test driver" program, which calls the module with a well-defined setup of parameters, registers, global data blocks, files, etc., and displays the output. Appendix A outlines guidelines for writing test drivers.

Testing a module in isolation also often requires a substitution of stub modules for calls to lower-level subroutines not yet written. Appendix B describes UTSTUB, a utility program in SOFTLIB for building FORTRAN stub modules.

A variety of utilities within the operating systems are usually heavily used during unit testing. These types of tools are suggested in Section 5.

4.1.7 Guidelines for Some Specific Environments

In developing specific types of program systems, the following general recommendations may be helpful as guidelines; individual tasks or projects should always consider their unique requirements.

Type of Module	Recommended Method to Perform Unit Testing	Coverage and Personnel
Data Aquisition or control	Inspections; isolation with unit test drivers	Use complete coverage only on the modules accessing the hardware; moderate coverage for others
Computational or numerical	Inspections or desk check; isolation with unit test drivers for bottom modules; unit test from calling program for top modules	Use complete coverage only on the modules performing numerical utilities and which are primarily computational; moderate coverage for others
User interaction, interaction, data entry, or editors	Structured walkthroughs; unit test from the calling program	Use independent people to unit test; the programmer should not test his/her own modules
Graphics	Inspections and structured walkthroughs; unit test from the calling program for the user-available functions; unit test with unit test drivers only if fundamental or primitive utility graphics are part of the system	Use independent people to unit test all user-available functions

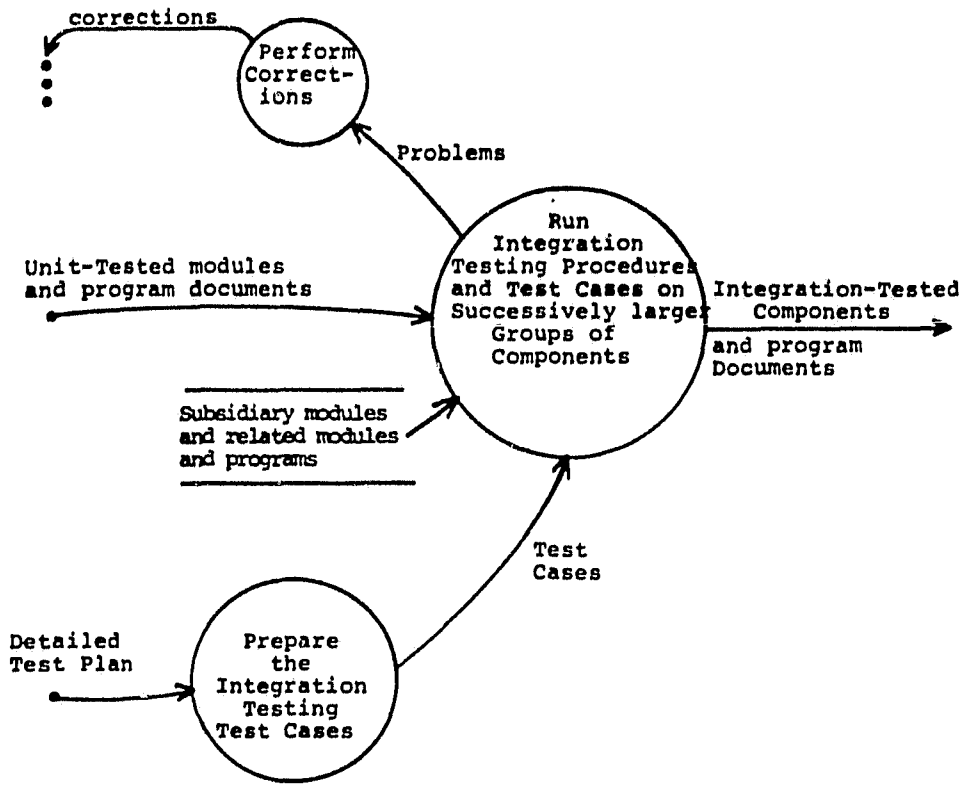
Type of Module	Recommended Method to Perform Unit Testing	Coverage and Personnel
Data basing and reporting	Inspections or desk checks; unit test from the calling program for the reporting and user-available functions; unit-test with test drivers for the data base access modules	Use complete coverage only on the modules accessing the data base; moderate coverage with independent people for others
Subsystems of a large project, where other pieces are missing	Inspections and structured walkthroughs; design and write a simulation of the missing pieces only if the schedule allows no other approach	Make the simulation available to other subsystem groups; involve them in the walkthroughs
System-level programs with heavy use of operating system features	Inspections or desk checks; isolation with unit test drivers for modules making the operating system service calls; unit test from calling program for others	Use independent people whenever possible; use complete coverage for the bottom modules, moderate or complete coverage for all others
A one-person task	Desk check; use other review methods and outside help on the most critical modules; use whichever unit test method is appropriate for the application	Involve the Task Requestor to unit test from top-down until all user interfaces have been checked

Unit testing is very susceptible to being overdone, and can be the largest expenditure of manpower in a project if the proper selections are not made and monitored. Despite expectations, if unit testing shows results much better or worse than expected, a manager should always be ready to modify the approach to be more efficient. Some recommendations to help the selections be cost effective include:

- Don't unit test to show that a program works; test to find its errors.
- Concentrate on the most critical parts and the most used parts.
- Involve independent people whenever possible.

- Don't unit test without a plan.
- Test results should be reproducible.
- Always inspect, review, or desk check the code.
- Be practical; don't insist on checking every path in every module.

4.2 A GENERAL SPECIFICATION FOR INTEGRATION TESTING OF COMPONENTS



4.2.1 Purpose

Component integration testing verifies the proper execution, logic, data, control flow, external functionality and interfaces in executable programs or groups of inter-related modules, isolated from other such groups. It is performed after the individual modules and subprograms have been verified. They are consolidated into functionally related packages or executable programs. These packages are first debugged as a whole, with no inputs, to establish that they can cycle. Then, controlled inputs are introduced to test the packages' ability to respond properly. Integration testing of components may proceed in several levels corresponding to successively larger groups of subprograms within the system.

4.2.2 Entrance Criteria

- All of the component modules have passed unit testing
- Documents associated with the prorams are up-to-date
- If the programs to be tested have logical dependencies on other modules and programs, they should have already passed integration testing
- The system hardware and software configuration is available
- The Detailed Test Plan for the programs is up-to-date
- Any performance measurement tools have been installed or designed and implemented.

4.2.3 Exit Criteria

- The programs successfully pass all specified integration tests without programmer intervention
- The program documentation has been updated, and agrees with the program operation
- If the project has a Librarian, the programs are signed off by the Librarian.

4.2.4 Personnel

This stage of testing is ordinarily performed by one or more of the software's programmers, or an independent team of programmers. In larger groups, the testing is usually led by the Chief Programmer in charge of the program.

4.2.5 Component Integration Testing Methods

4.2.5.1 General Description

The steps in component integration testing are generally as follows:

(1) A sequence in which modules are to be combined into testable components is selected.

(2) Test cases are prepared for component integration testing of these components.

(3) The group of modules with the necessary stubs or test drivers is constructed to isolate a subset of the program.

(4) The component integration testing is conducted by running test cases of data and parameters on the subset.

(5) Corrections are made to the modules and their documentation, and they are re-tested.

(6) The stubs are replaced with successively lower-level modules, OR successively higher-level modules are added underneath the test drivers, and further test cases are run. This is continued until an executable program has been completely integrated.

(7) The executable program is tested with whatever commands or command procedures the user will be employing.

(8) For very large multi-program systems, the same procedures are conducted with successively larger combinations of executable programs, verifying their interfaces with test cases.

(9) Maintain test logs; obtain signoff from librarian or designated personnel.

4.2.5.2 Methods

In this stage of testing, the unit-tested modules are combined in successively larger packages until an executable program is fully-tested. This may often require integrating higher-level modules together with "stubs", or dummy lower-level modules. In integration testing a program or set of modules, the most effective sequence for packaging and testing is not the same for all programs. The best choice depends on the type of environment with which the software must interact. The three types of approaches from which to select are:

- Top-down
  Test from the top (main program) down (to the lower modules) for programs which are primarily interactive. (E.g., menus or graphics.) This allows user sequences to be experimented with and critiqued as soon as possible.

- Bottom-up
  Test from the bottom modules up for programs which make heavy or critical use of hardware devices, or which make heavy or critical use of operating system services. (E.g., a data acquisition program or a program which relies on network services.) In addition, programs which have any critical I/O performance criteria are also best tested bottom-up. (E.g., programs doing heavy disk I/O, data base access, or primitive graphics.) This method allows bottlenecks and critical I/O to be tested first, allowing as much time as possible if redesign is required.

- Sandwich testing
  Test both the highest- and lowest-level modules first for programs which have an approximately uniform mixture of interface types (e.g., some user interaction, some data base I/O) with no one type being critical or overwhelming. Then continue testing modules successively further from both the top and the bottom. Computational programs should generally be tested this way. This method allows the most commonly used types of programming data structures and data passing techniques to be evaluated with the least risk of late re-structuring efforts.

The program's application and structure should be used to select one of these three test sequences for each separate program or set of modules. In larger systems, this selection should be made during design activities, and written into the program's Test Plan.

4.2.5.3 Guidelines for Selecting Component Integration Test Cases

The test cases used in component integration testing are similar to those listed in Section 4.1.5.3 for unit testing. However, since unit testing is already complete, the emphasis should be on testing module interfaces. Thus, the test cases should be composed of (refer to the checklist, Section 4.1.5.3, for details):

Class 1. and Class 2.	Parameter Limits and Parameter Partitioning	--	Test the calls to those modules which were newly added to the test group.
Class 3.	Return Codes	--	Test the handling of all codes returned from the modules newly added to the group.
Class 7.	Linkages	--	Try to cause each newly-added module to call modules which it references.

The other test case classes should be included as appropriate. Interactive, graphic display, or report generation applications should include testing by one or more end-users to evaluate the user interfaces top-down.

4.2.6 Tools for Component Integration Testing

The most useful tools in component integration testing are the "test driver" and "stub module generator" programs described for use in unit testing, Section 4.1.6.

In multi-tasking environments, component integration testing additionally requires using "stub tasks". A stub task is a dummy executable program which can run or be run by another program for testing inter-program interfaces. The details of a stub task depend on the operating system and environment it must simulate, but they are generally very simple programs.

As an example under RSX 11M using FORTRAN, if RESUMing of a second task is to be tested, a useful stub task may look like:

```
      PROGRAM xxxxxx
C
C     STUB TASK FOR INTEGRATION-TESTING INTER-TASK "RESUME"
C
        DIMENSION NAME(3)
        DATA NAME/'xx','xx','xx'/
C
C     WRITE BRIEF MESSAGE AND SUSPEND IMMEDIATELY
C
 10     WRITE (1,1) NAME
 1      FORMAT (' ***>',3A2)
        CALL SUSPEND
C
C     IF RESUMED, JUST SUSPEND AGAIN
C
        GO TO 10
C
        END
```

This stub can now serve as a task for a program to RUN; if it is RUN first by the human tester, it can serve as a task for a program to RESUME. If desired, the tester can ABORT the stub task when it first SUSPENDS. More sophisticated stubs can easily be conceived, such as stubs which set, clear, or wait for event flags.

Although this example is specific to RSX-11M, it is clear that stub tasks can be very simple. They should be kept simple, wherever possible, as a multi-tasking environment cannot be well-represented by such tasks without substantial simulation work.

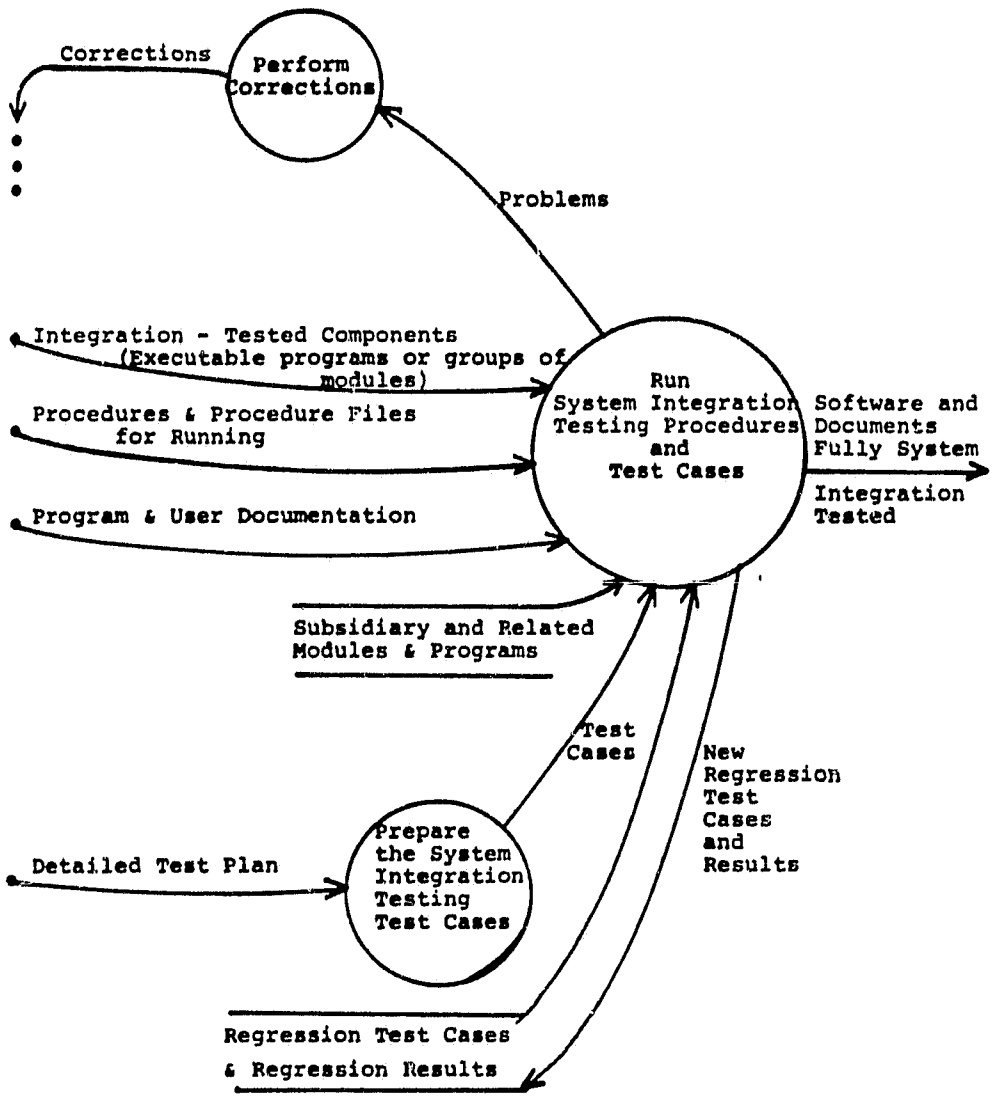
Typical operating system utilities useful in integration testing are suggested in Section 5.

4.2.7 Guidelines for Some Specific Environments

In developing specific types of program systems, the following general recommendations may be helpful as guidelines; individual tasks or projects should always consider their unique requirements.

Type of System or Environment	Recommended Sequence in which to Perform Component Integration Testing
Data acquisition or control	Bottom-up
Computational or numerical	Sandwich
User interaction, data entry, or editors	Top-down
Graphics	Top-down or Sandwich
Data basing and reporting	Sandwich
Subsystems of a large-project, where other pieces are missing	Sequence depends on the application
System-level programs with heavy use of operating system features	Sandwich or bottom-up
A one-person task	Sequence depends on the application

4.3 A GENERAL SPECIFICATION FOR SYSTEM INTEGRATION TESTING



4.3.1 Purpose

System integration testing detects interface problems and operational, functional, and logical problems when a complete software system is operated as a whole. It also detects problems in the operating, program, and user documentation. For modifications to existing software, system integration testing also verifies that unchanged software and functions still operate as they used to.

4.3.2 Entrance Criteria

- All components (executable programs or groups of modules) have passed component integration testing
- Operating, program, and user documentation are up-to-date and available
- Procedure files and written procedures for the operation of the system have been prepared
- The Detailed Test Plan for the system is up-to-date
- Any test cases for this phase of testing that have passed detailed design are being prepared
- Any test cases and their results from the previous system release are available ("Regression Test Cases")
- If the system has any logical dependencies on other programs or modules, they should have passed all stages of testing
- The system hardware and software configuration is available
- Any performance measurement tools have been installed or designed and implemented

4.3.3 Exit Criteria

- The system properly initiates, operates, and terminates all functions, using the procedure files or written procedures
- Successive executions of the software generate the proper number of versions of output files, and remove all temporary files upon exit.
- The system proceeds properly through all test cases prepared for system integration testing, generating

the correct results without programmer intervention

- All test cases from the previous system release ("regression test cases") execute properly and produce the expected results
- Program, operator, and user documents are all updated

4.3.4 Personnel

The most effective approach to system integration testing is to employ a team which is all or partly made up of persons who did not develop the new or changed software. An independent test group (programmers and users who did not develop the system) is best. Members of the development team who were not specifically involved in developing the release's new functions are also a good choice. One or more users or operators are important to this type of testing. If the task or project is not large enough to have any of these types of independent programmers to do the system integration testing, it is best to (1) try to involve users or the Task Requestors, and (2) concentrate on coverage detail in writing the test plan, test cases, and expected results well before this phase of testing begins.

Regardless of the team's makeup, it is best for the experience level to be a mixture of junior and senior programmer staff. The senior personnel are needed to effectively test and debug all interfaces, while the junior staff should organize and run the test cases and procedure under supervision. Both junior and senior programming staff should observe users, in order to evaluate the system and correct its documentation.

4.3.5 System Integration Testing Methods

4.3.5.1 General Description

The steps in system integration testing are generally as follow:

(1) A sequence and overall approach is selected.

(2) Test cases specified in the Detailed Test Plan for system integration testing are prepared, including any regression tests.

(3) The system components are combined into successively larger builds of executable programs, and the test cases re-executed for each. Corrections are made and re-testing is done.

(4) Regression tests and other tests are run on the entire

system.

(5) Corrections are made to the programs and documents, and re-testing is done.

(6) Maintain test logs and/or problem reports; obtain signoff from librarian or designated personnel.

4.3.5.2 Methods

Several effective methods can be used in this phase of testing. As in all phases of testing, some of the methods are most effective for specific system environment. However, some or all of the following approaches will be highly effective in each individual environment.

- Incremental integration
- Functional testing
- Regression testing

Incremental Integration - For medium or large size systems, putting all of the software together at once and trying to test it just doesn't work. To be effective, systems composed of components of multiple executable programs, or those composed of one very large program (with over 50 modules or so) should be system integration tested in several successively more complete "builds" of the system components. In these environments, system integration testing is a smooth transition from component integration testing, as shown in the overview diagram in Section 1.2. When executable program components are built and tested in these successively larger combinations, it is important that the plan for what will be included in each of the builds be carefully written into the Detailed Test Plan ahead of time. For the first "skeleton" build and for each successive build, appropriate test cases specified in the Test Plan should be prepared and executed. The overall sequence in which the components are collected into builds should attempt to integrate the most critical functions earliest (following the same guidelines as outlined in Section 4.2.5 for component integration testing). If more than "a few" successive builds are planned, the test cases and results for each should be organized into "regression tests", as described below.

Regression Testing - This is the most powerful method of system integration testing for the following environments:

- A system which will be tested with "incremental integration", as described above, and where the number of successive builds is not small. Regression methods should be used if each successive build introduces new software and features which may possibly impact the software and features already tested in the previous

build.

- A system which will undergo several or many releases during its lifetime. Regression methods should be used unless each release introduces only new, unrelated software or functions, and each release does not require rebuild or reconfiguration of the previous release's software.

Regression testing consists of preparing test cases which can be run with little or no change on two different versions of the system; the results are compared to determine and analyze any differences. In regression testing during "incremental integration", test cases are run on two successive builds, and the results are compared to verify that the software and functions of the previous build were not affected by the new components. In regression testing of successive releases of a system, test cases are run on two successive releases, and the results are compared to verify that the "unchanged" portions of the system are still correct. In either case, the tests verify that the remainder of the system has not "regressed", or lost its correctness, when other small subsections are changed.

Regression testing can be done by manually running the programs and comparing the results. It is much more efficient if the test case data can be saved in files for easy running, and if the results can be saved in files for easy comparison. Not all systems lend themselves to this semi-automation, but the files should be saved wherever practical.

Since the test cases will be run on two slightly different systems, some of the data will be incompatible, or will produce results which are rightfully incompatible. The individual test cases used for regression testing should be chosen to segment the system's functions and software, with each test case covering a subset of the system.

Programmers should examine all discrepancies and resolve them. "Incremental integration" should not proceed until regression testing of one build against its previous one is entirely successful. If any corrections are required to the software or the test cases during regression testing, they should all be re-run.

When system integration testing is complete, all test cases used for the current release should be evaluated as likely candidates for saving as regression tests to be used in the next release. The regression test plan saves significant test time in most system environments, and the regression test plans and test cases should be updated whenever possible.

Regression tests should be designed and maintained to be easily

run whenever maintenance changes are made on a system, even if the changes are "small". The regression tests can be used by maintenance programmers to check their work even before a set of changes are collected into a release.

Functional Testing - To verify that a system (or build) performs all (or a subset) of its required functions and external specifications, a specially-directed set of tests should be used. This is especially important in systems with a variety of external functions, or systems with many users and generally not operated by the developers. Functional Testing should be performed by a team with one or more users, operators or the Task Requestors. Test cases should be written as a checklist which parallels the superset of: (1) the system's complete, written functional specifications, and (2) the system's User's Guide or Operating Instructions. If incremental integration is being performed, the functional tests associated with each successive build should be specified. Any error in the system's performance should be recorded and analyzed separately by the development programmers; any discrepancies in the user or operator documents should be corrected during the functional testing. If corrections are made to any parts of the software, all of the functional tests should be repeated.

4.3.5.3 Guidelines for Creating System Integration Test Cases

The following areas should all be included, using the suggestions in Section 4.1.5.3 for unit tests, in order to design the actual test cases:

- All inter-program control paths should be executed at least once in each possible mode.
- Inter-program data transfer paths should be tested, with partitioning and limit-tests similar to those suggested in Section 4.1.5.3 for unit tests.
- All inputs and outputs should be tested or verified; the recovery of the system from a missing or bad input should be checked.
- "Isolated failure testing" should be performed for all devices upon which the system depends (See Appendix C).
- "Volume testing" should be performed, testing small, medium, and large quantities of data and numbers of users.
- Keeping each test restricted to a given function allows the results to clearly identify problems encountered.

- Performance should be tested to verify the system's performance under physical limitations, system loads, data throughput stress, etc.

- All interfaces to the operating personnel should be tested for recovery from poor data, clarity of message, and actual performance as documented.

- All diagnostic features of the system should be tested.

- All features specified in the system's functional requirements should be organized in a checklist and Test Case Matrix with enough coverage to test each feature.

- Verify that documentation is sufficient for Acceptance Testing and user and operator training.

4.3.6 Tools for System Integration Testing

System integration testing may require familiarity with many of the operating system utilities suggested in Section 5. In addition, configuration control is an important element of this stage of testing. The following special approaches may be instrumental to efficient system integration testing:

- Naming conventions and version identification conventions -- These are crucial to uniformly identify test cases and results files; they are absolutely essential for unequivocal identification of module versions and successive builds of executable programs and the whole system.

- Disk utilization conventions -- Separate storage locations for components, test cases, and each separate build or release of the system.

- Software librarian -- On larger projects, a central software librarian should specialize in enforcing the above conventions, and in maintaining the files and monitoring or controlling their access.

- Command procedures -- Running of the test cases (as well as comparisons) are considerably more efficient if catalogued procedures can be written for repeated operations. This is critical to avoid unrepeatable situations caused by "pilot error". Even short sequences of commands should be saved as catalogued procedures whenever possible.

Comparisons of results for regression testing are always easier whenever results can be captured onto files in any form.

Comparison methods include:

- Save ASCII files, print them, compare them manually - this may be the only way for reports which vary or which are informally structured.
- Use file compare utilities for formatted ASCII files.
- Dump unformatted files. Most dump utilities can be setup to output their dump to an ASCII file. Unformatted files can then be compared by dumping them to ASCII files, and then using standard file compare utilities on the ASCII dumps.

4.3.7 Guidelines for Some Specific Environments

In working in certain system environments, the following general recommendations may be useful as guidelines; individual tasks or projects should always consider their unique requirements.

Testing Approaches

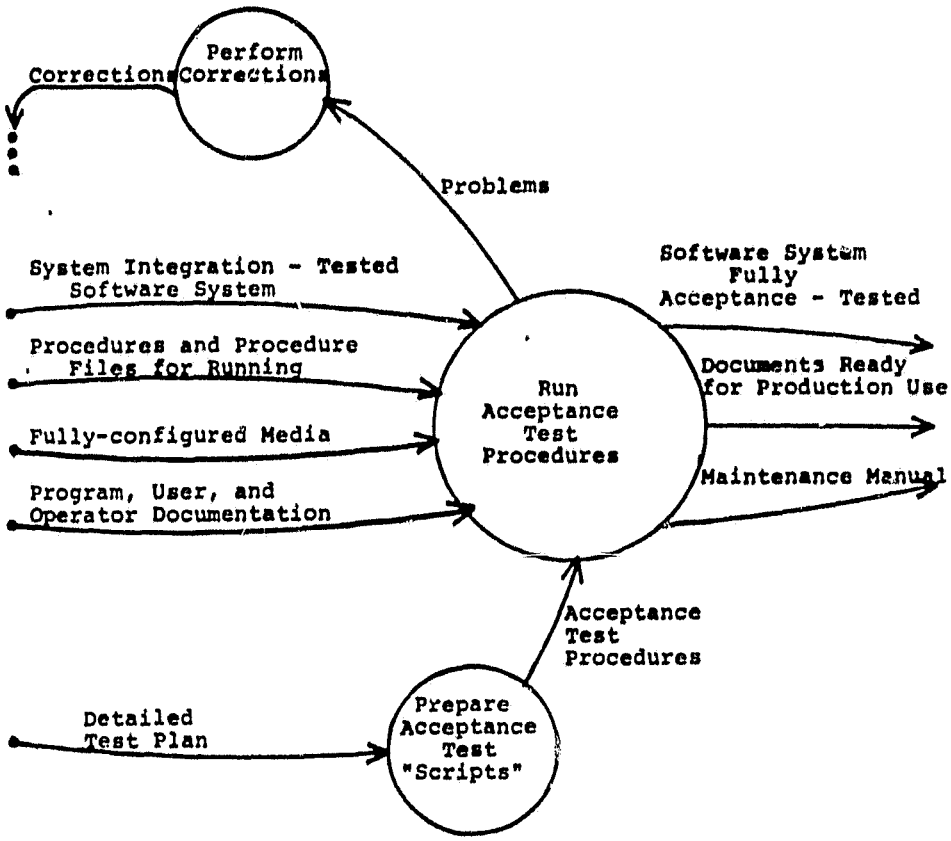
Long-lifetime systems	Rely heavily on semi-automated Regression Testing; add new test cases to regression tests after each release; perform Functional Testing after all other tests.
Interactive systems	Write a full checklist of Functional Testing test cases which match the User's Guide or Operating Instructions; users, operators, or the Task Requestors should perform the Functional Testing.
Computational or data basing applications	Rely heavily on semi-automated Regression Testing; save data base files for comparisons.
Transaction systems or editors	Save the data base or edit file before and after performing operations which are specified in written test case procedures.

System-level programs with heavy use of operating system features (or heavy use of a package)	Regression Test cases should be prepared and saved, even if the software is not expected to undergo continued change, because the software will require repeated System Integration Testing again each time the operating system (or package) undergoes a new release. It should never be assumed that a manufacturer will leave any functions unchanged from one operating system release to another.
A one-person task	Use the most appropriate methods, but involve the Task Requestor for help in Functional Testing; alternatively, obtain approval to temporarily utilize some personnel from outside the task for Functional Testing.

Configuration Control and Tools

Unix systems	Use the Source Code Control System to carefully manage version identification and control of files. Use pipelines to capture the input and output of programs onto files.
Virtual systems (VMS, VM, etc.)	Use system features to store and capture program input and output, even for interactive programs.
RSX and VMS systems	Do not rely on the file version number for distinguishing changes; use separate directories (UICs) and subdirectories for various builds and releases.
Micro and Mini Systems	Whenever possible, store separate system versions for testing on separate disk volumes; however, do not allow any changes to be made anywhere except on one single designated volume.

4.4 A GENERAL SPECIFICATION FOR ACCEPTANCE TESTING



4.4.1 Purpose

Acceptance tests are tests used to demonstrate that the computer program and its documentation perform according to the performance and technical requirements in its functional specification, and is ready to use in its intended mission. From the viewpoint of the user, these are the most important tests in the testing process, although the research nature of many Ames tasks occasionally introduce some informality at this stage. Overall system functionality is completely verified; in many cases, contractual requirements are also accepted at this stage.

4.4.2 Entry Criteria

- The system has passed all portions of system integration testing.
- Operating, program, and user documentation are up-to-date and available.
- Procedure files and written procedures for the operation of the system are up-to-date and available.
- The system has been prepared on the proper media ordinarily used for its production environment, including any related packages or utilities.
- The proper user accounts have been setup on the system.
- The Detailed Test Plan for the system is up-to-date.
- Any acceptance test procedures and test cases are being prepared into written test "scripts" to be followed.
- The system hardware and software configuration is available.
- Any performance measurement tools have been installed and tested.
- Schedules for user participation, user training, and system cutover have been prepared and approved, and the required personnel are available.
- Full documentation from hardware, operating system, and package vendors is available, as well as any other related Government-furnished documentation.

4.4.3 Exit Criteria

- The system operates satisfactorily in all of the planned test procedures according to the test scripts and user documentation.
- Successive executions of the software leave a "clean" disk configuration.
- Successive executions of the test scripts and repetitions by different users generate repeatable behavior and results.
- Performance factors have been measured to be satisfactory for system release.
- Program, operator, and user documents are all updated.
- A maintenance manual has been prepared or updated for the system.
- Any improvements suggested by the acceptance test personnel have been recorded, categorized, discussed with the Task Requestor, and either resolved or scheduled for future releases.

4.4.4 Personnel

Acceptance testing is performed by one or more of:

- Users
- Operators
- Task Requestor
- Maintenance personnel
- Other personnel designated by the Task Requestor.

Program development personnel should participate in the testing only to help record problems and later analyze or correct them.

4.4.5 Acceptance Testing Methods

4.4.5.1 General Description

The steps generally involved in acceptance test are:

(1) The development team, in conjunction with users and the Task Requestor, prepare written procedures and their expected results for a number of specific system-level functions to be acceptance-tested. These procedures are written into test "scripts" detailed enough for a user to follow.

(2) The users to be involved in acceptance testing are

trained to the extent that future new users will be.

(3) Acceptance test should then begin with an evaluation of the development testing history of the software, to allow the Task Requestor to verify that proper internals and "correctness" testing has been performed in previous testing phases.

(4) Packaging should then be reviewed by verifying media disk directories, sizes, files for load procedures and document contents.

(5) On-line testing should then proceed using some functional test cases.

(6) User-oriented testing should then be performed by allowing user-level personnel to carry out, in their own style, a number of procedural scripts which have been prepared in advance.

(7) Additional methods, as outlined below, are executed.

(8) Maintain problem reports and test logs; obtain signoff from librarian and designated personnel.

4.4.5.2 Methods

One or more of the following methods should be used in acceptance testing.

Functional Test Cases - As for system integration testing, test cases to test selected external functions can be prepared and run by the users, following the appropriate documentation. This method is desirable for selectively verifying the new functions in successive releases of a long-lifetime system. The functional test cases should be specified in the Detailed Test Plan, and should follow the system's functional specifications. All reports and screens should be included in these test cases, to ensure final review.

Procedural Scripts - Users with varying levels of familiarity should use the final documentation to follow the test procedures which have been written into test "scripts". Although the test scripts should specify what functions to perform, how to do them, and what to expect, the users should be allowed to interpret the scripts, in the context of the user documentation, in their own style. The amount of such testing should be proportional to both the number of separate functions in the system and the number of individual users.

Unstructured Testing - As a substitute for, or in addition to procedural scripts, users can be invited to perform random

testing of the system features based on documentation. Personnel with varying levels of familiarity should be included, in order to detect the types of errors that would affect novices, experts, and maintenance programmers. Some scenarios should still be written in this method, in order to achieve the full coverage which procedural scripts do. This is the best approach for final testing of system security on projects where security is an important feature.

Parallel Operation Test (cutover) - A system can be setup to be used redundantly with a predecessor system or a previous release for a certain period of time. The appropriate personnel should still operate both systems, which may generate a higher demand on the operators. Although this approach clearly accomplishes unstructured testing, some functional test cases and procedural scripts should also be used in cases where the circumstances during the parallel operation do not cover all testable functions. In addition, evaluation and comparison procedures and tools should be written ahead of time in the Detailed Test Plan, to make sure the new system gets properly evaluated.

Pilot Operation Testing - A specific production environment can be setup for the first use of a system for a certain period of time. This pilot environment can be a low-risk production use or a repeat of an already-completed production run. As with parallel operation testing, some functional test cases, procedural scripts, comparison procedures, and comparison tools should also be used to make sure that all testable functions are properly covered.

Benchmark Testing - A benchmark is a set of specific test cases which are run at acceptance test time to measure and evaluate a system's performance factors. Some benchmarks are written to be additionally run on a totally different system for comparison, but in all cases the performance factors to be measured, as well as their acceptable ranges, should be completely specified along with the test cases in the Detailed Test Plan. Benchmarks used for acceptance test are similar to system integration testing test cases, except they should be formulated primarily by users and the Task Requestor. To ensure that the test is valid, a Code Inspection should be held with both the preparer and the designer, after the design from the Detailed Test Plan is prepared (programs coded and/or data prepared).

In addition to the above methods, measurements should be performed during acceptance testing both to verify the system's acceptability and to provide a baseline against which to test changes. Always measure wall-clock performance, storage usage, memory and other resource usage.

4.4.6 Tools for Acceptance Testing

Since acceptance testing is performed primarily by users, elaborate tools should not be part of this stage; the system being tested, or its documentation, should provide all that is necessary to perform the testing. The configuration control approaches outlined in Section 4.3.6 for system integration testing should be carefully observed by all participants; programmers and users should be especially careful to avoid modifying any system components without the appropriate reporting, approval, and retesting.

Performance measurement aspects of acceptance testing may require some special performance monitoring and measurement tools. These are almost always specialized to the operating system or language being used, but include elements such as:

- Overall system monitors (E.g., "SHOW" and "MONITOR" on VAX/VMS; "RMDEMO" on RSX-11M; "METER" on RTE);
- Accounting statistics maintained by multi-user operating systems;
- Approximate cpu-time and wall clock time measurements which can be requested at the command language level in most operating systems, for measuring a program by subtracting "before" from "after";
- Timing calls which can be inserted into the program's source code (usually language-dependent). When using these types of calls, be careful to account for whatever the smallest time-able increment of time is on the machine being used. Also, report all timing data outside of all portions of the program which are being timed; the I/O for reporting is always a significant slice of time.

4.4.7 Guidelines for [illegible] Environments

The following general recommendations should be used as guidelines; each individual task or project should always consider their unique requirements. Checked combinations are recommended, but all approaches should be considered.

TYPE OF APPLICATION	Functional Test Cases	Procedural Scripts	Unstructured Testing	Parallel Operation Testing	Pilot Operation Testing	Benchmark Testing	Notes
Data Acquisition or Control	X	X	X	X	X		Perform extensive timing and performance measurement
Computational or Numerical	X			X	X	X	Perform timing measurements if relevant
User interaction, data entry, or editors	X	X	X		X		On multi-user systems, test under a variety of system loads
Graphics	X	X	X	X	X	X	Strong configuration control is recommended
Data basing and reporting	X	X		X	X		Benchmark and time if relevant
System-level programs, with heavy use of operating system features	X	X	X	X	X		Benchmark and time if relevant; test under a variety of loads
A one-person task	X	X	X	X	X	X	All, as appropriate ; use Task Requestor or approved outside personnel for user testing
Long lifetime systems	(Concentrate on testing of the newly-released features)		X				

Section 5
TOOLS AND REPORTING

5.1 OPERATING SYSTEM FEATURES

Testing requires more knowledge and use of operating system features and utilities than any other programming activity. Programmers involved in testing should learn these features and utilities ahead of time. Because of their importance, the most experienced programmers should also be part of the test team or available for providing help. The most used features include:

- On-line debuggers: E.g.: SDB on UNIX; VAX/VMS FORTRAN Debuggers; RSX-11M ODT and XDT; RSX-11M BUG (Macro 11 deassembler and debugger being submitted to SOFTLIB); debuggers for microprocessor development systems.

- Compiler options: E.g.: optional-compile debug lines; FORTRAN TRACE options; conditional assembly debug lines: "p" option in C compiler.

- System monitors: E.g.: SHOW, MONITOR, ACCOUNTING on VAX/VMS; RMDEMO, ATL, TAS, TAL on RSX 11M; "w", "ps-al", "dsd" on UNIX; METER on RTE; job accounting statistics on batch systems.

- File dump utilities

- File compare utilities (usually only useful for ASCII files, but unformatted files can be "dumped" into ASCII files, and their dumps compared.)

- Source control systems: E.g.: SCCS and RCS under Unix

- Operating systems designed for debugging: E.g.: Motorola's EXORMACS; Intel's MDS.

- File modification utilities: E.g.: ZAP on RSX-11M

5.2 SPECIALIZED TEST TOOLS

Unit testing and component integration testing are aided by having a stub module generator, which can quickly create a dummy stub module to act for any arbitrary module. SOFTLIB contains UTSTUB, a stub module generator for FORTRAN environments. (See

Appendix B for a description.)

Unit test drivers are more complicated and need to be rewritten for each test situation. Guidelines for such a driver are in Appendix A.

Stub tasks are commonly used for integration testing in multitasking environments. An example is shown in Section 4.2.6.

Testing in the multitasking environment of RSX-11M when using global event flags (or testing of a VAX/VMS program using global event flags) may be aided by the use of EFL, a SOFTLIB utility providing a terminal user full access to all global event flags.

Writing new test tools may often seem desirable. Test tools, however, require substantial investments of time in order to be generally useful. It is imperative that management permission always be obtained before embarking on the development of any test tool, no matter how appealing. Project schedules must always account for these activities. SOFTLIB should always be checked for applicable testing tools, which are continually being collected there.

5.3 REPORTING OF TEST ACTIVITIES

Which test cases are run on each software component version, and what the results are, is the primary data which should be recorded to document testing. The following approaches are recommended:

Keep a test log to record which test cases were run, the software component and version, personnel, summary of results, and time and date. A test log should be kept by each "tester", even if the task environment does not explicitly request it. Projects with a librarian should centralize the test logs when testing is performed by teams.

Keep the test plan up-to-date. It should be modified each time a new release is planned.

Keep the test cases and their results (on-line whenever possible). This is imperative at the system integration testing phase, as the test cases are likely to be maintained for regression testing.

Report problems on written problem reports. This approach should be used whenver the personnel performing testing are different from the developers. Even on very small or one-person tasks, the user or Task Requestor will perform some acceptance testing; written software problem reports should be used at that time. In most cases, the problem report is much more important than any suggested correction. Since other portions of the software may be

simultaneously under test by other personnel, the problem report must be manually used to determine the best correction which is consistent with all other corrections being made.

Section 6
REFERENCES

The two best texts on testing:

- Myers, Glenford, The Art of Software Testing, Wiley; 1979.
- Myers, Glenford, Software Reliability, Wiley; 1976.

NASA/Ames reference on Inspections:

- Guidelines for Software Inspections, Informatics General Corporation NASA Contractor Report to Ames Research Center; Nov. 25, 1983.

Other references on testing and related methodology:

- Deutsch, Michael, Software Verification and Validation, Prentice-Hall; 1982.
- Yourdon, Edward, Managing the System Life Cycle, Yourdon Press; 1982.
- Draft Standard for Software Test Documentation, IEEE Inc., 1982.
- Boehm, Barry, "Software and Its Impact", Datamation; May, 1973

APPENDIX A

GUIDELINES FOR UNIT TEST DRIVERS

Appendix A
Outline for Unit Test Drivers

A test driver calls an isolated module for testing in a controlled environment. The driver should explicitly guarantee the values of parameters, registers, common blocks, files, etc., and show all outputs from the module. It should allow the programmers to easily modify parameter values in order to cover various test cases.

It is not necessary for a test driver to be interactive. It is often more desirable for it to be non-interactive, in order to be simpler and thereby decrease the possibility of introducing artificial errors through the test driver. Re-compiling a test driver for each test case is, for many modules, a very viable approach. Simplicity and quick development are absolute standards to be applied in writing test drivers.

Test drivers over 3 pages of code long should not be written, as this indicates either the module itself is overly complex, or that the test driver is too sophisticated, too interactive, or too generalized.

Test drivers ordinarily are better if highly specific to the module being tested. If a number of related or similar modules will be written, however, a generalized test driver may be justified.

A skeleton for possible unit test drivers follows:

(1)
- READ parameters to be passed to the module under test
- READ two control parameters: (A) whether or not to stop and (B) whether or not to print parameters
- IF (parameter (A) requested to stop) THEN exit with a message
- IF (parameter (B) requested to print) THEN echo all parameters to output
- CALL the module under test, passing the parameters
- IF (parameter (B) requested to print) THEN print all parameters to output again

- REPEAT starting at (1) for next test case.

Unit test drivers should generally be constructed so that the test cases of input parameter sets can be read from stored files.

APPENDIX B
SAMPLE FORTRAN STUB MODULE GENERATOR (UTSTUB)

Appendix B
Sample FORTRAN Stub Module Generator (UTSTUB)

UTSTUB is an interactive program which produces a dummy FORTRAN stub module useful for top-down testing purposes. It runs under both RSX-11M and VAX/VMS, but can be easily ported to other environments. The UTSTUB program is available from SOFTLIB.

IF UTSTUB is installed in an RSX-11M or VAX/VMS environment, it is invoked by entering

```
>STUB ABCDEF
```

to build a stub for module ABCDEF. In other environments, it is simply run. In either case, UTSTUB will prompt for the module's name if it is missing from the command line or illegible.

Once invoked, UTSTUB prompts for the number of calling parameters and the type of each parameter. Parameters may be single variables or arrays.

UTSTUB also can provide a "debug" option. If selected, this option causes the generated stub modules to print "CALL TO MODULE ABCDEF", plus all values of that module's calling parameters. If no "debugging" is requested, the stub produced simply returns.

APPENDIX C
TECHNICAL MEMO: ISOLATED FAILURE MODE TESTING

Interoffice Memo

Informatics Inc

To: SWTS Group

From: Russ Molari
Location: Palo Alto

Date: June 2, 1980
cc:

Subject: Isolated Failure Mode Testing

This memo describes a specific, simple testing technique which I believe is particularly effective when used during "integration testing" - that is, the phase of testing when software and hardware are tested together, or portions of independently-developed software are tested together.

There are often a large number of possible failure modes of an integrated system. Some failure modes are simple and result from a single unacceptable condition (e.g. - hardware turned off, program not installed, illegal parameter value). Some failure modes may result from a complex of unacceptable conditions (e.g. - an iterative computation which does not converge, disk hardware which is producing undetected read errors). Some failure modes result from internal software problems or bugs (e.g. - opening an improperly constructed file name, writing an incorrect number of words per I/O block).

Some failure modes are particularly easy to isolate, and can be directly investigated during integration testing. The most useful to consider in this brief scheme are those failure modes of an integrated system which are caused by a single unacceptable condition over which the user, operator, or attendant has direct control. Examples are:

- A required piece of hardware is turned off.
- A required piece of hardware has an incorrect switch setting.
- A required piece of hardware is not ready or is off-line for a specific reason (such as a missing write-ring on a tape).
- A required system resource is not available (such as memory).
- Some required software is missing.
- The required result of a prior program is missing (such as a file expected from a prior program).
- Some input was not provided.

These types of cases may conceivably be quickly corrected by the attendant, and the failing program resumed or re-run.

A technique for isolating and testing this category of failure modes can be simply performed, as follows:

- Determine the hardware and software components on which the program relies and over which the attendant has some direct control.
- Determine the components' corresponding unacceptable conditions which can occur and which the attendant may have caused and/or may be able to correct.
- Cause each of these situations to occur in separate runs.
- Observe the behavior of the program and correct and improve it as much as possible to provide highly meaningful user-oriented messages and/or a clean recovery.

As an example, consider a hypothetical tape output program. The program relies on being able to write data onto a magnetic tape. In one test run, the program is run with the tape drive turned off. A resulting abort and message ("open status return = -9 on unit 1") can be improved to give the operator a second chance with a better message ("The tape drive is off. Please turn it on and type GO to continue.") In a second test run, the tape is mounted without a write-ring. A resulting abort and message ("I/O status return = -82 on unit 1") can be improved to give the operator another chance with a better message ("The tape is missing a write-ring. Please re-mount it with a ring and type GO to continue.") In a third test run, the tape drive is set to the wrong parity or density. It is discovered that the program continues and writes the tape anyway; the programmer must then correct the program or the documentation.

This example indicates the underlying philosophy of this type of isolated failure mode testing: namely, anticipate, cause and test the easy and simple failure modes. The improvements will be very worthwhile, as they will cover the common areas in which operator errors affect the program. Deliberately causing these failures during testing allows the programmer an easy, direct way of evaluating program behavior and error codes which might otherwise have been totally based on a hardware manual. This will also circumvent having a programmer puzzle over error codes at a later, more inconvenient time, trying frantically to work in the opposite direction (such as trying to quickly look up error codes such as "-82").

This approach to testing can be effective for software components, also. For example, consider an application where one program runs a second one. The programmer can and should deliberately cause test failure modes such as: the second

program is not available or installed; the first program produces output data used by the second, and it is missing; the second program is already active; etc.

For these reasons, I recommend including this type of deliberate "isolated failure mode" testing whenever you do integrated testing.

COMMENTS

AMES DOCUMENT # 356 Date_____

NAME_____ ORG CODE_____

Phone_____ Mail Stop_____

REQUESTS/COMMENTS

Return to CDDC
M/S 233-13